Ch. 18: ADTs: Stacks and Queues
18.1 and 18.4

CS 2308
Spring 2015

Jill Seaman

Abstract Data Type

- A data type for which:
  - only the properties of the data and the operations to be performed on the data are specific,
  - not concerned with how the data will be represented or how the operations will be implemented.
- In fact, an ADT may be implemented by various specific data types or data structures, in many ways and in many programming languages.
- Examples:
  - ProductInventory (impl'd using array **and** linked list)
  - string class (not sure how it's implemented)

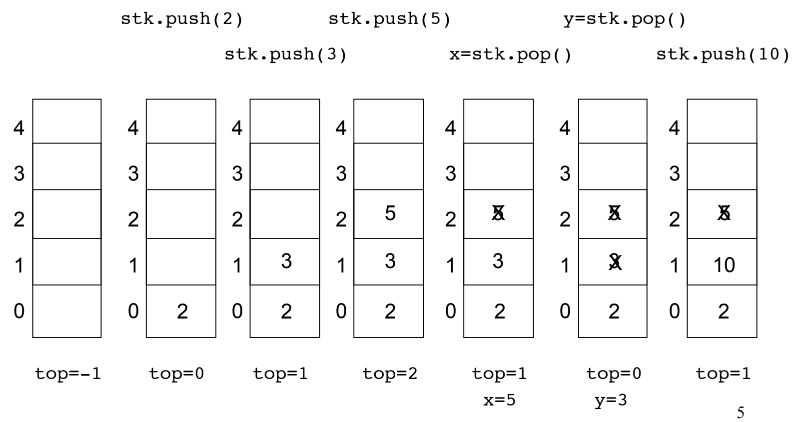
Introduction to the Stack

- Stack: an abstract data type that holds a collection of elements of the same type.
  - The elements are accessed according to LIFO order: last in, first out
  - No random access to other elements
- Examples:
  - plates or trays in a cafeteria
  - bangles . . .

Stack Operations

- Operations:
  - push: add a value onto the top of the stack
    - make sure it's not full first.
  - pop: remove a value from the top of the stack
    - make sure it's not empty first.
  - isFull: true if the stack is currently full, i.e.,has no more space to hold additional elements
  - isEmpty: true if the stack currently contains no elements

Stack illustrated



Stack Applications

- Execution of programs
  - return address: location of statement following the function call
  - When a function is called, the return address and the local variables are stored on a stack.
  - When the function terminates, the local variables are removed from the stack and the return address is retrieved.
- Parsing: (i.e. matching brackets like: (), {}, [])
- Evaluating arithmetic expressions in post-fix notation. (i.e. 4 5 + 7 2 - * is (4+5)*(7 - 2) = 45)

Implementing a Stack Class

- IntStack:
  - contains ints
  - implemented using an array of ints of a fixed size
- Alternative implementations of an integer stack:
  - use a dynamically allocated array of ints, resize when it becomes full
  - use a linked list with nodes that contain ints (see 18.2)
  - std::stack from the C++ library (STL) (see 18.3)

IntStack: A stack class

```
class IntStack
{
private:
   static const int STACK_SIZE = 100; // The stack size
   int stackArray[STACK_SIZE];        // The stack array
   int top;                  // Index to the top of the stack

public:
   // Constructor
   IntStack();

  // Stack operations
   void push(int);
   int pop();
   bool isFull() const;
   bool isEmpty() const;
};
```

IntStack: constructor

```
//***********************************************
// Constructor                                  *
// This constructor creates an empty stack.     *
//***********************************************

IntStack::IntStack()
{
  top = -1;                       // empty
}
```


IntStack: push

```
//***************************************************
// Member function push pushes the argument onto   *
// the stack.                                       *
//***************************************************

void IntStack::push(int num)
{
   assert (!isFull());

   top++;
   stackArray[top] = num;

}
```

stack overflow

if (!isFull()) is false, the program will exit with an error message.

IntStack: pop

```
//*****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and returns it as the result.   *
//*****************************************************

int IntStack::pop()
{
  assert (!isEmpty());

  int num = stackArray[top];
  top--;
  return num;
}
```

stack underflow

if (!isEmpty()) is false, the program will exit with an error message.

IntStack: test functions

```
//***************************************************
// Member function isFull returns true if the stack *
// is full, or false otherwise.                     *
//***************************************************

bool IntStack::isFull() const
{
   return (top == STACK_SIZE - 1);
}

//****************************************************
// Member function isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//****************************************************

bool IntStack::isEmpty() const
{
   return (top == -1);
}
```

IntStack: driver

```
#include<iostream>
using namespace std;

#include "IntStack.h"


int main() {

    // set up the stack
    IntStack stack;
    stack.push(2);
    stack.push(3);
    stack.push(5);
    int x;
    x = stack.pop();
    x = stack.pop();
    stack.push(10);
    cout << x << endl;

}
```

What is output?

What is left on the stack when the driver is done?

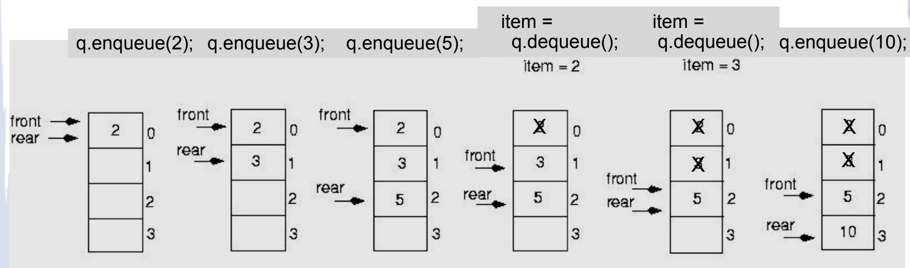
Introduction to the Queue

- Queue: an abstract data type that holds a collection of elements of the same type.
  - The elements are accessed according to FIFO order: first in, first out
  - No random access to other elements
- Examples:
  - people in line at a theatre box office
  - print jobs sent to a (shared) printer

Queue Operations

- Operations:
  - enqueue: add a value onto the rear of the queue (the end of the line)
    - make sure it's not full first.
  - dequeue: remove a value from the front of the queue (the front of the line) "Next!"
    - make sure it's not empty first.
  - isFull: true if the queue is currently full, i.e.,has no more space to hold additional elements
  - isEmpty: true if the queue currently contains no elements

Queue illustrated



Note: front and rear are variables used by the implementation to carry out the operations

```
int item;
q.enqueue(2):
q.enqueue(3);
q.enqueue(5);
item = q.dequeue(); //item is 2
item = q.dequeue(); //item is 3
q.enqueue(10);
```

Queue Applications

- The best example applications of queues involve multiple users or processes:
- On a shared printer a queue is used to hold print jobs submitted by users of the system, while the printer services those jobs one at a time.
  - Submitting a print job => enqueue
  - Printing the next job => dequeue
- Communications software uses queues to hold data received over networks.
  - When data is transmitted to a system faster than it can be processed it is placed in a queue as it is received.

Implementing a Queue Class

- IntQueue:
  - contains ints
  - implemented using an array of ints of a fixed size
- Alternative implementations of an integer queue:
  - use a dynamically allocated array of ints, resize when it becomes full
  - use a linked list with nodes that contain ints (see 18.5)
  - std::deque and std::queue from the C++ library (STL) (see 18.6)

Implementing a Queue class
issues using a fixed length array

- The previous illustration assumed we were using an array to implement the queue
- When an item was dequeued, the items were NOT shifted up to fill the slot vacated by dequeued item
  - why not?
- Instead, both front and rear indices move through the array.

Implementing a Queue Class

- When front and rear indices move in the array:
  - problem: rear hits end of array quickly
  - solution: wrap index around to front of array

					7	9	6
					front		rear

q.enqueue(3):

3					7	9	6
rear					front		

q.enqueue(4):

3	4				7	9	6
	rear				front		

Implementing a Queue Class

- To “wrap” the rear index back to the front of the array, you can use this code to increment rear during enqueue:

```
if (rear == queueSize-1)
   rear = 0;
else
   rear = rear+1;
```

- The following code is equivalent, but shorter (assuming 0 <= rear < queueSize):

```
rear = (rear + 1) % queueSize;
```

- Do the same for advancing the front index.

Implementing a Queue Class

- When is it full?

3	4				7	9	6
	rear				front		

```
q.enqueue(5);
q.enqueue(2);
q.enqueue(1);
```

Note: enqueue increments rear

3	4	5	2	1	7	9	6
				rear	front		

- It’s full:

(rear+1)%queueSize==front

Implementing a Queue Class

- When is it empty?

```
int x;
for (int i=0; i<queueSize;i++)
   x = q.dequeue();
```

Note: dequeue increments front

after the first one:

3	4	5	2	1		9	6
				rear		front	

one element left:

				1			
				front rear			

no elements left, front passes rear:

				rear	front		

- It’s empty:

(rear+1)%queueSize==front

Implementing a Queue Class

- When is it full? (rear+1)%queueSize==front
- When is it empty? (rear+1)%queueSize==front
- How do we define isFull and isEmpty?
  - Use a counter variable, numItems, to keep track of the total number of items in the queue.
- enqueue: numItems++
- dequeue: numItems--
- isEmpty is true when numItems == 0
- isFull is true when numItems == queueSize

IntQueue: a queue class

```
class IntQueue
{
private:
   static const int QUEUE_SIZE = 100; //The queue size
   int queueArray[QUEUE_SIZE];        // The queue array
   int front;        // Subscript of the front elem
   int rear;         // Subscript of the rear elem
   int numItems;     // Number of items in the queue

public:
   // Constructor
   IntQueue;

  // Queue operations
   void enqueue(int);
   int dequeue();
   bool isEmpty();
   bool isFull();
};
```


IntQueue: functions

```
//*******************************************************
// Creates an empty queue                               *
//*******************************************************

IntQueue::IntQueue()
{
   front = 0;      //where the first elem will go
   rear = -1;      //advanced before adding elems
   numItems = 0;   //total number of elems in the queue
}
```


IntQueue: enqueue

```
//********************************************************
// Enqueue inserts a value at the rear of the queue.   *
//********************************************************

void IntQueue::enqueue(int num)
{
   assert (!isFull());

   // Calculate the new rear position
   rear = (rear + 1) % QUEUE_SIZE;

   // Insert new item
   queueArray[rear] = num;

   // Update item count
   numItems++;
}
```


IntQueue: dequeue

```
//*******************************************************
// Dequeue removes the value at the front of the      *
// queue and returns it.                              *
//*******************************************************

int IntQueue::dequeue()
{
   assert (!isEmpty());

   // Retrieve the front item
   int num = queueArray[front];

   // Move front
   front = (front + 1) % QUEUE_SIZE;

   // Update item count
   numItems—;

   return num;
}
```

IntQueue: test functions

```
//*******************************************************
// isEmpty returns true if the queue is empty,          *
// otherwise false.                                      *
//*******************************************************

bool IntQueue::isEmpty()
{
   return (numItems == 0);
}

//*******************************************************
// isFull returns true if the queue is full, otherwise *
// false.                                                *
//*******************************************************

bool IntQueue::isFull()
{
   return (numItems == QUEUE_SIZE);
}
```


IntQueue: driver

```
#include<iostream>
using namespace std;

#include "IntQueue.h"

int main() {

   // set up the queue
    IntQueue q;
    int item;
    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(5);
    item = q.dequeue();
    item = q.dequeue();
    q.enqueue(10);
    cout << item << endl;

}
```

What is output?

What is left on the queue when the driver is done?